Alma Mater Studiorum · Università di Bologna

Corso di Laurea in Ingegneria e Scienze Informatiche

Studio delle prestazioni del simulatore Alchemist: ottimizzazione di routing e caching

Relatore:
Prof. Mirko Viroli

Presentata da:
Giacomo Scaparrotti

Correlatore:
Ing. Danilo Pianini

Prima Sessione di Laurea
Anno Accademico 2016/2017

a tutti coloro che,
in questi anni,
mi hanno sostenuto

Sommario

Il simulatore Alchemist è un simulatore ad eventi discreti (DES) che fa dell'efficienza uno dei suoi punti di forza, grazie anche all'adozione di una versione modificata del Next Reaction Method di Gibson e Bruck e di altre soluzioni incentrate sull'ottenimento di elevate prestazioni. Dopo aver allestito un banco di prova con cui testare il simulatore in differenti situazioni, però, è emerso che alcune soluzioni tecniche adottate, come il ricorso al caching per cercare di ridurre al minimo le elaborazioni computazionalmente più impegnative, non contribuivano in maniera adeguata al mantenimento di un alto livello prestazionale. Si è perciò cercato di porre rimedio a questi problemi, introducendo delle nuove soluzioni tecniche e migliorando alcune di quelle preesistenti, misurando e discutendo poi il loro impatto sulle performance di Alchemist.

La seguente trattazione è organizzata come segue: nel capitolo 1 si introduce il simulatore Alchemist, spiegandone i princìpi di funzionamento; nel capitolo capitolo 2 se ne approfondisce l'architettura, mostrandone anche il lato tecnico; nel capitolo 3 si mostrano invece le performance che Alchemist era in grado di offrire prima dell'introduzione delle nuove soluzioni tecniche, che sono poi illustrate, con le relative misurazioni, nel capitolo 4.

Indice

1 Background

1.1 Introduzione ad Alchemist

Alchemist [14] è un simulatore ad eventi discreti (DES) [3, 9], ossia un simulatore in cui gli eventi vengono eseguiti uno alla volta avanzando il tempo di simulazione conseguentemente[12]. Questo tipo di approccio consente di modellare con facilità eventi collocati nel tempo con le più svariate distribuzioni temporali. Gli scenari tipici possono riguardare i più diversi fenomeni, dalle reazioni chimiche[13] al movimento dei pedoni[18, 5]: è sufficiente poter esprimere il fenomeno da simulare nei termini del modello di Alchemist. Questo modello è composto di entità astratte ben distinte, la cui concretizzazione è demandata alla specifica categoria di simulazioni, ossia alla specifica incarnazione.

Alchemist è derivato da un modello prettamente chimico; di conseguenza, la terminologia utilizzata per indicare le varie entità all'interno di Alchemist si rifà a quella solitamente utilizzata in chimica, come vedremo tra poco.

1.2 Il modello computazionale di Alchemist

Ci sono varie entità all'interno di Alchemist, riportate di seguito andando dal generale al particolare:

Environment L'Environment è l'ambiente all'interno del quale si svolgono i fenomeni oggetto della simulazione.

Node Un Node è un'entità che si trova all'interno dell'ambiente. Ogni nodo è una sorta di contenitore all'interno del quale sono presenti ulteriori entità, in particolare:

Molecule Una Molecule è il nome di un particolare dato contenuto all'interno di un nodo. Un Node può contenere un numero arbitrario di molecole.

Concentration Una Concentration è il valore associato al dato espresso da una molecola. Una concentrazione è quindi associata ad una molecola in particolare; si possono paragonare una molecola e una concentrazione ad una variabile in un linguaggio di programmazione e al valore ad essa associato. Il tipo di dato di una concentrazione è generico: a diversi tipi di dato corrispondono diverse 'incarnazioni' del simulatore.

Reaction Una Reaction è un qualsiasi evento che può cambiare lo stato dell'ambiente. Ogni nodo contiene un insieme (anche vuoto) di reazioni, le quali sono a loro volta costituite da un insieme di azioni, che modellano nel concreto il modo in cui la reazione agisce sull'ambiente, ad esempio muovendo un nodo al suo interno o cambinando il valore di una concentrazione, da una serie di condizioni che devono essere soddisfatte affinchè la reazione possa avvenire, e da una distribuzione temporale, che determina sostanzialmente la frequenza con cui una reazione ha luogo, tenendo conto naturalmente che affinchè una reazione possa svolgersi devono essere soddisfatte tutte le condizioni.

Neighborhood Il Neighborhood (letteralmente 'vicinato') è un'entità composta da un nodo centrale e da un insieme di altri nodi che costituiscono il vicinato. Ogni nodo ha il proprio vicinato, che può essere anche vuoto. Il vicinato di un nodo può cambiare con l'avanzare della simulazione, aggiugendo ed eliminando nodi.

Linking Rule Il criterio con cui si determina se un nodo debba far parte del vicinato di un altro nodo o meno è determinato da una Linking Rule, in base allo stato corrente dell'ambiente, seguendo una regola che può essere definita arbitrariamente.

Figura 1.1: Rappresentazione grafica delle entità di Alchemist

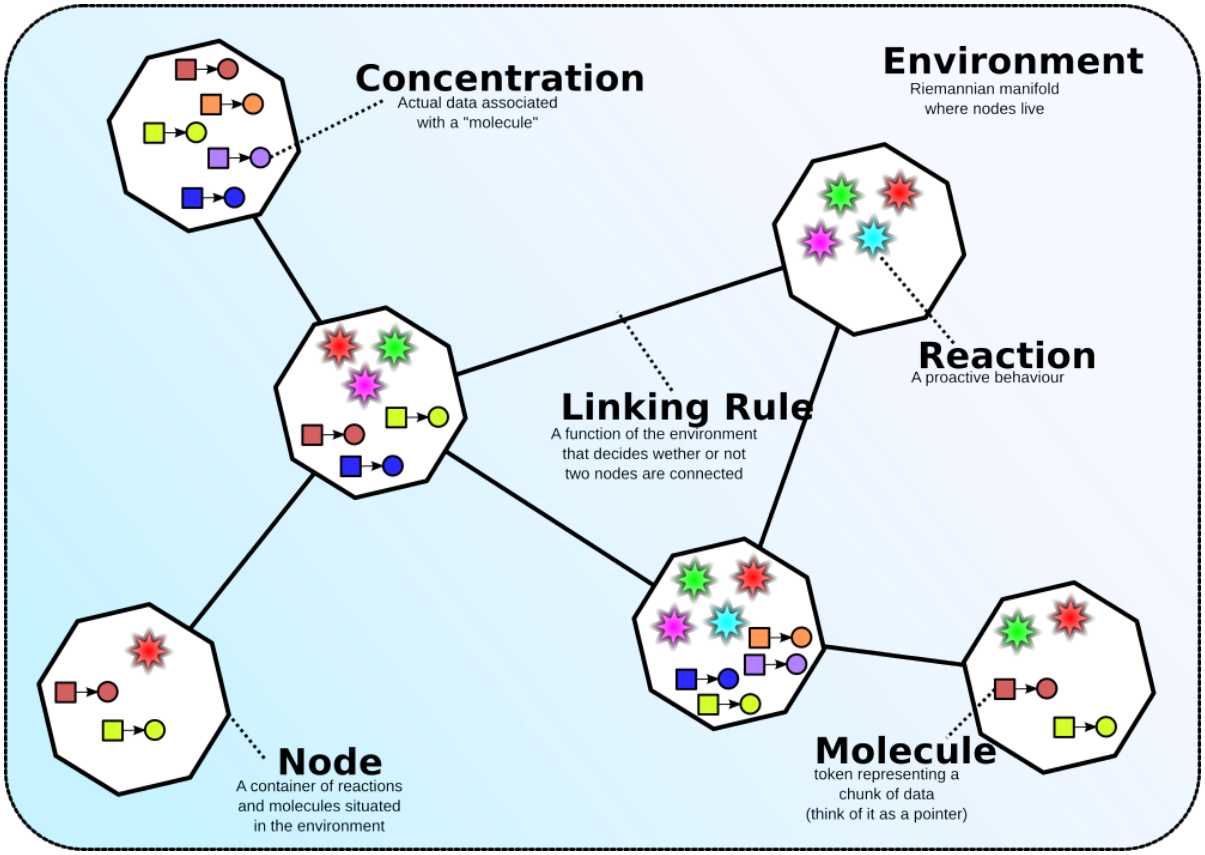
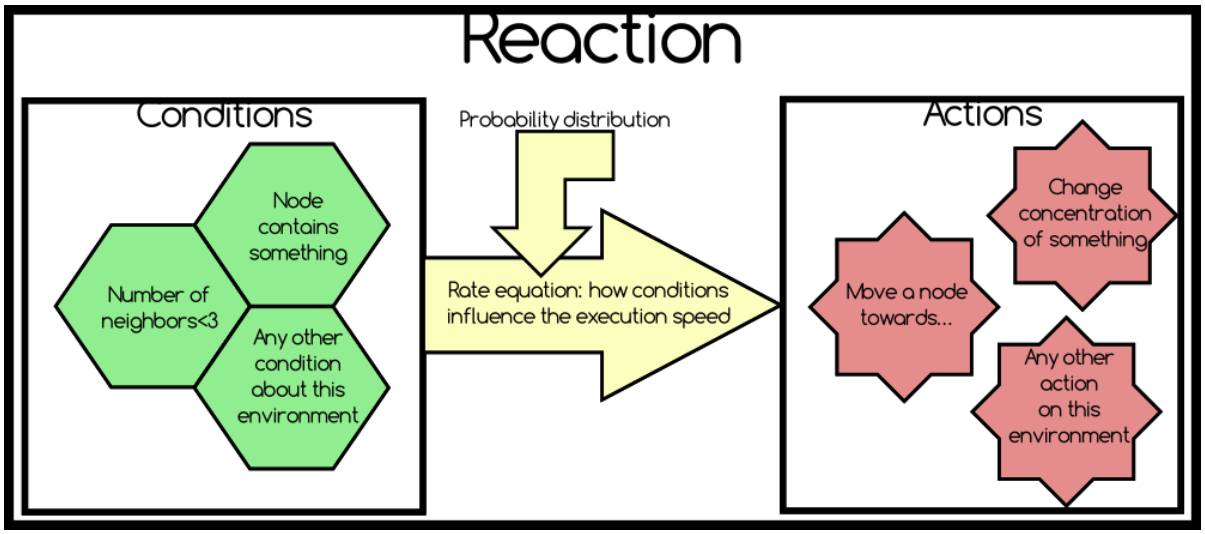


Figura 1.2: Rappresentazione grafica dello svolgimento di una Reaction



In Alchemist le entità del dominio possono essere concretizzate in maniere anche molto differenti: sulla base di ciò, è stato introdotto il concetto di Incarnation, o incarnazione, che è una specializzazione del modello di Alchemist che aiuta a definire specifici tipi di simulazione, specificando le implementazioni concrete delle entità di Alchemist per una certa categoria di simulazioni, operando di fatto una traduzione model-to-model (M2M).

1.3 Interfacciamento con Protelis

Protelis[15] è un linguaggio di programmazione, basato sul field calculus[17] e derivato dal linguaggio Proto [4], studiato appositamente per la scrittura di programmi da eseguire su aggregati di dispositivi.

Protelis è un linguaggio interpretato la cui Virtual Machine è scritta in Java, e tra le altre cose questa prevede che esista un Execution Context, che è un'interfaccia tra il programma Protelis e l'ambiente all'interno del quale il dispositivo che lo sta eseguendo si trova. Il contesto d'esecuzione fornisce al programma accesso alle informazioni del dispositivo in modo analogo alle classi in Java System e Java Runtime.

Come detto precedentemente, ogni incarnazione fornisce una definizione di concentrazione; nel caso dell'incarnazione Protelis, una concentrazione è definita come avente tipo `Java.lang.Object`, ossia è possibile manipolare qualsiasi oggetto Java. I nodi Alchemist hanno la possibilità di eseguire programmi scritti in questo linguaggio, che vengono trattati come reazioni e, di conseguenza, schedulati ed eseguiti come tali; un programma scritto in Protelis ha la possibilità di interagire con le varie entità di Alchemist, compresi i vari nodi nell'ambiente e le relative molecole e concentrazioni; inoltre, è possibile per un programma Protelis in esecuzione 'dentro' un nodo avere accesso a quelle informazioni che avrebbe a disposizione anche il singolo nodo, ad esempio la propria posizione nell'ambiente. Queste funzionalità sono gestite dal contesto di esecuzione.

1.3.1 Il contesto di esecuzione

Ad ogni nodo che esegue un programma Protelis è associato un contesto di esecuzione. Questo ha il compito di fare da tramite tra il nodo e l'ambiente, tenendo traccia dello stato persistente del nodo, del suo stato in relazione a quello dei suoi vicini e dello stato del dispositivo in relazione all'ambiente. Nel particolare ambito di Alchemist (Protelis

è infatti un linguaggio a sè stante, e Alchemist è solo uno dei contesti in cui può essere utilizzato), il contesto di esecuzione fornisce accesso agli oggetti che costituiscono l'ambiente, permette di valutare la posizione del nodo, la distanza dai nodi che costituiscono il vicinato e l'accesso al vicinato stesso. Relativamente a questi ultimi due aspetti, è stato fatto un approfondimento dal punto di vista delle prestazioni, che verrà discusso nei prossimi capitoli.

1.4 Utilizzo di Alchemist

Alchemist può essere utilizzato in due diverse modalità: interattiva e batch. La differenza sta nel fatto che in modalità interattiva viene eseguita una sola simulazione per volta, mentre in modalità batch possono essere eseguite più simulazioni con una sola invocazione di Alchemist.

In entrambi i casi, la simulazione viene configurata mediante un documento in formato YAML[7], all'interno del quale, utilizzando gli opportuni tag, è possibile stabilire quanti nodi disporre nell'ambiente, in che posizione e quali molecole debbano contenere. Andando più nello specifico, è possibile specificare quale debba essere il tipo di Environment da utilizzare, con quale criterio i nodi debbano essere collegati tra di loro per formare un Neighborhood e quale sia il modo in cui la posizione di un nodo all'interno dell'ambiente viene espressa. Per ogni reazione viene specificata la relativa distribuzione temporale. Infine, è possibile specificare, come già visto, l'incarnazione.

All'interno del file YAML di configurazione è inoltre possibile definire delle variabili, aventi ciascuna sia un valore di default che un ulteriore range di possibili valori. In modalità batch un sottoinsieme di queste variabili viene utilizzato per generare una serie di simulazioni differenti, utilizzando ad ogni iterazione un valore all'interno del range di valori ammissibili e il valore di default per tutte le altre, mentre in modalità interattiva vengono utilizzati unicamente i valori di default per tutte le variabili. In entrambi i casi, il risultato di una simulazione viene restituito mediante i cosiddetti `OutputMonitor` ed `Exporter`: i primi vengono perlopiù utilizzati per registrare i dati della simulazione in corso, e per loro costruzione sono in grado di mostrare informazioni anche molto approfondite (anche l'interfaccia grafica è un `OutputMonitor`), mentre gli `Exporter` sono usati perlopiù per registrare metadati sulla simulazione; questa distinzione comunque non è netta, e spetta all'operatore utilizzare uno strumento piuttosto che un altro a seconda delle sue necessità.

1.4.1 Simulazione di esempio

Quello che segue è un semplice esempio di un file YAML per la configurazione di una simulazione:

```
1  incarnation: protelis
2
3  network-model:
4    type: EuclideanDistance
5    parameters: [0.5]
6
7  gradient: &gradient
8    - time-distribution: 1
9      program: >
10       def aFunction() {
11         1
12       }
13       aFunction() * self.nextRandomDouble()
14   - program: send
15
16 displacements:
17   - in:
18       type: Circle
19       parameters: [10000, 0, 0, 10]
20     programs:
21       - *gradient
```

Nel codice riportato si è scelto di utilizzare l'incarnazione Protelis (riga 1), e di collegare tra di loro i nodi facenti parte di un vicinato sulla base della loro distanza euclidea (righe 3 e 4); si può notare che oltre al tipo è definito anche un insieme di parametri (in questo caso composto da un solo parametro): questi saranno passati al costruttore della classe `EuclideanDistance` nel momento in cui questa verrà instanziata. È inoltre stato inserito un piccolo programma Protelis direttamente inline (righe 9-13); viene inoltre fatto riferimento al programma send (riga 14), il cui scopo è quello di inviare a tutti i nodi facenti parte del vicinato di quello che lo esegue il risultato delle sue computazioni. È stata poi definita la disposizione dei nodi e quali programmi debbano essere da loro

eseguiti (in questo caso, quello definito poco al di sopra): 10000 nodi vengono disposti all'interno di un cerchio di raggio 10 centrato nel punto [0,0] (righe 16-19).

Ciò che viene realizzato da questa piccola simulazione è molto semplice: tutti i nodi posizionati inizialmente rimangono fermi e, man mano che la simulazione avanza, la concentrazione della molecola contenuta al loro interno cambia in maniera casuale; questa simulazione potrebbe proseguire indeterminatamente.

Naturalmente è possibile realizzare file YAML più ricchi per la realizzazione di simulazioni più complesse: tutto dipende da quale fenomeno si ha intenzione di studiare. Inoltre, è possibile (ed è in effetti ciò che si fa nella maggior parte dei casi reali) inserire il codice Protelis in un file separato a cui fare riferimento dall'interno della simulazione, in maniera tale da eseguire i programmi che saranno poi distribuiti per l'esecuzione nel mondo reale.

2

Architettura e design di Alchemist

2.1 Introduzione

Alchemist è un software realizzato principalmente in Java, sfruttando appieno le funzionalità avanzate del linguaggio. Sono inoltre utilizzati, in misura minore, anche altri linguaggi Java-compatibili, come Scala e Xtend, che comunque, essendo eseguiti anch'essi all'interno della JVM, non introducono problemi di compatibilità: Alchemist risulta perciò eseguibile su qualsiasi macchina per la quale sia disponibile la Java Virtual Machine.

All'interno di Alchemist è molto rigida l'applicazione dei principi di buona programmazione. È inoltre applicato il pattern Model-View-Controller nella sua forma Entity-Control-Boundary, dove il modello è costituito dall'implementazione delle entità precedentemente descritte, il controller dal cosiddetto Engine e da tutte le classi correlate, ed il boundary dagli OutputMonitor e dagli Exporter.

2.2 Il motore

Il motore di Alchemist è implementato principalmente nella classe Engine, e rappresenta il controller nel contesto del pattern ECB. Al motore spetta il compito di far avanzare la simulazione, scandendone i diversi momenti, e di gestire tutte le reazioni la cui esecuzione è stata programmata ma non ancora effettivamente svolta. Quest'ultimo aspetto è gestito da una versione estesa del cosiddetto Next Reaction Method di Gibson e Bruck[10, 11], che permette ad Alchemist di scalare bene all'aumentare del numero di eventi da simulare: all'interno del motore viene mantenuto un grafo orientato che gestisce le dipendenze

tra le reazioni, così da sapere quale sia la successiva reazione che può e deve essere eseguita in base a quali altre reazioni dipendono da essa e devono quindi attenderne il completamento.

Ogni volta che un nodo all'interno dell'ambiente o il suo vicinato subiscono un cambiamento - ad esempio un nodo viene spostato o viene aggiunto al vicinato di un altro nodo - questo cambiamento viene notificato al motore, in maniera tale da mantenere sempre aggiornato il grafo delle dipendenze.

Inoltre, tra gli eventi interni al simulatore gestiti direttamente dal motore c'è la gestione del tempo della simulazione, e l'evasione dei cosiddetti comandi, ossia operazioni eseguite dal motore che possono riguardare tutti gli eventi da esso gestibili (ad esempio, la messa in pausa in pausa della simulazione).

2.3 L'interfaccia Environment

Un Environment, o ambiente, è lo spazio all'interno del quale avviene la simulazione. La definizione di ambiente è astratta, lasciando piena libertà al progettista della simulazione per quanto riguarda la definizione del numero di dimensioni dell'ambiente, delle sue dimensioni, della sua forma. Si possono quindi avere, ad esempio, ambienti quadrati bidimensionali così come ambienti sferici tridimensionali con una superficie curva, ostacoli e metriche non-euclidee.

Un ambiente fornisce gli strumenti per determinare la posizione di un nodo al suo interno e la distanza tra due nodi, nonchè per muovere i nodi e per ottenere informazioni su di essi (ad esempio riguardo le molecole che contengono). Inoltre, sono forniti anche gli strumenti per determinare il criterio con cui due nodi vicini sono considerati tali.

Naturalmente, essendo un'interfaccia, Environment fornisce solo le signature dei metodi che permettono di svolgere queste operazioni, mentre l'implementazione spetta alle sue sottoclassi.

2.3.1 OSMEnvironment: un approfondimento

OSMEnvironment è un'implementazione dell'interfaccia Environment che modella un ambiente costituito da una mappa sulla quale è possibile muovere i nodi calcolando il percorso da seguire tramite la libreria GraphHopper. Questa è una libreria open source studiata appositamente per calcolare i percorsi da seguire per andare da un punto

ad un altro su una mappa, ed è una delle librerie open source più performanti nel suo genere. Per il suo funzionamento può sfruttare le mappe messe a disposizione da OpenStreetMaps, le quali sono a loro volta open source. Gran parte dei metodi introdotti da OSMEnvironment in aggiunta a quelli delle classi da cui eredita sono dedicati proprio alla navigazione, o routing.

Un nodo che si trova all'interno di un OSMEnvironment può sia seguire una traccia GPS precedentemente fornita che muoversi in base ad un percorso calcolato; è altresì possibile far muovere un nodo in base ad entrambi questi criteri, seguendo l'uno o l'altro a seconda delle circostanze. Si può decidere, mediante un'opportuna configurazione del file YAML, quale debba essere la frequenza con cui ricalcolare il percorso da seguire: questa operazione è infatti un'azione che compone una reazione e, come visto nel capitolo precedente, ad ogni reazione è associata una distribuzione temporale che determina la frequenza con cui la reazione avviene. Dal momento che la logica per il movimento dei nodi si trova all'interno di OSMEnvironment, ciò che viene compiuto dall'azione consiste nell'invocare il metodo di OSMEnvironment, di cui l'azione ha un riferimento, che restituisce il percorso da seguire per raggiungere il punto desiderato.

L'esistenza di OSMEnvironment permette di realizzare simulazioni dove i nodi si muovono su una mappa del mondo reale, seguendo strade e sentieri realmente esistenti. Nel prossimo capitolo verrà discusso l'aspetto prestazionale di queste operazioni.

2.4 Il routing in Alchemist

Il routing dei nodi sulla mappa è una funzionalità presente all'interno di Alchemist in diverse declinazioni: è possibile chiaramente realizzare un simulazione che consiste nel muovere i nodi su una rete di strade, ma è possibile anche avere dei vicinati in cui i collegamenti tra i nodi vicini sono realizzati seguendo dei percorsi invece che essere semplicimente in linea d'aria, così come è possibile determinare la distanza tra due vicini in base a quella che il primo dovrebbe percorrere sulle strade per raggiungere il secondo. Queste tre applicazioni del routing sono indipendenti tra di loro e possono perciò essere combinate a piacimento, a seconda del risultato che si vuole ottenere.

Esperimenti svolti tramite Alchemist in cui si è fatto ricorso alla possibilità della navigazione sulla mappa sono presenti anche in letteratura, e uno di questi[16], consistente nello steering dinamico delle folle nella città di Vienna, basa il suo funzionamento proprio

su questa funzione, sia per muovere i nodi sulla mappa che per valutare la distanza tra i vicini.

L'implementazione del routing all'interno delle simulazioni porta con sè alcune problematiche relativamente al costo computazionale che comporta:l'individuazione del percorso più breve tra due punti su una mappa è, in fin dei conti, un'istanza particolare del problema dei cammini minimi in un grafo, problema a cui nel tempo sono state trovate numerose soluzioni, tutte accomunate dall'avere, in varie misure, delle complessità asintotiche polinomiali[8]. Poichè non è noto un modo per ridurre la complessità di questi algoritmi, è necessario far sì che il loro utilizzo sia il più misurato possibile, in modo da non introdurre dei costi evitabili nell'esecuzione delle simulazioni: l'analisi di questo problema sarà uno dei punti focali dei capitoli successivi.

3

Analisi delle prestazioni di Alchemist

In generale, Alchemist ha adottato fin dalla sua nascita delle soluzioni volte a mantenere un elevato livello prestazionale. La scelta di utilizzare il Next Reaction Method di Gibson e Bruck va esattamente nella direzione di ottimizzare le prestazioni prima di tutto sul piano concettuale, ossia degli algoritmi utilizzati, e solo successivamente su quello del raffinamento del codice.

Risulta comunque interessante studiare alcuni degli algoritmi utilizzati all'interno di Alchemist, per poi cercare, se necessario, di migliorare la loro implementazione nel tentativo di aumentare le prestazioni.

3.1 OSMEnvironment e GraphHopper

Ci sono degli aspetti relativi alle performance di Alchemist che più di altri si prestano ad essere analizzati in maniera approfondita; il routing è uno di questi.

Come visto in precedenza, OSMEnvironment mette a disposizione la possibilità di realizzare simulazioni in cui i nodi si muovono sulle strade del mondo reale, guidandoli con l'ausilio della libreria GraphHopper.

Di default, OSMEnvironment utilizza per la navigazione l'implementazione fornita da GraphHopper dell'algoritmo di Dijkstra[8], che ha una complessità computazionale di $O(|E| + |V| \log |V|)$. In linea di principio, questo fa sì che, in presenza di un numero di nodi molto elevato, il tempo di esecuzione tenda a salire rapidamente. Conseguentemente, quando è stata implementata la classe OSMEnvironment, è stato riservata attenzione a questo aspetto: ogni volta che è necessario ottenere un nuovo percorso, prima di ricorrere a GraphHopper per il suo calcolo, lo si cerca all'interno di una cache presente

all'interno di ogni istanza di OSMEnvironment. Se il percorso è già stato calcolato in precedenza, non sarà necessario calcolarlo nuovamente, poichè sarà già presente nella cache. Quest'ultima è implementata tramite la classe LoadingCache, presente all'interno della libreria Google Guava[6], la cui implementazione è quindi collaudata grazie ad un ampio utilizzo.

Al fine di valutare con precisione le prestazioni della cache è stato implementato un banco di prova all'interno di Alchemist che permette di fare due importanti operazioni. La prima consiste nel misurare, passo per passo, il tempo 'reale' trascorso dall'inizio della simulazione, in maniera tale da poter quantificare con precisione sia il tempo necessario allo svolgimento di una simulazione, sia come questo cambi al variare dei parametri della simulazione (ad esempio, la frequenza con cui viene ricalcolato il percorso che ciascun nodo deve seguire). L'altra operazione consiste nel quantificare il cache hit rate della Guava Cache utilizzata all'interno di OSMEnvironment, in modo da valutarne in maniera oggettiva l'efficacia. Quest'ultima operazione ha richiesto anche la modifica dello stesso OSMEnvironment: la cache è infatti un oggetto privato all'interno dell'ambiente, che originariamente non prevedeva la possibilità di ottenere da essa questa informazione: bisogna infatti considerare che l'effettuazione di misure di questo tipo impatta negativamente sulle prestazioni della cache, poichè la misurazione di questo parametro ritarda necessariamente l'evasione delle richieste in arrivo alla cache, perciò non può e non deve essere eseguita costantemente, ma solo quando realmente necessario.

In entrambi i casi, l'esportazione di queste informazioni è stata effettuata costruendo degli appositi Exporter, i quali, avendo accesso sia all'ambiente che all'Engine ed essendo invocati ogni volta che viene terminato un passo della simulazione, rappresentano la scelta ideale per questo tipo di utilizzo.

Al fine di rendere più attendibili possibile i risultati ottenuti, le simulazioni oggetto di benchmark vengono eseguite all'interno di un solo thread; in questo modo, si evita di aggiungere l'ulteriore elemento di aleatorietà costituito dallo scheduling del sistema operativo con la relativa prelazione dei thread.

Sono perciò innanzitutto state eseguite delle misurazioni sulle prestazioni del calcolo delle route e sull'impatto della cache; gli esperimenti sono stati effettuati sulla base della simulazione presentata nell'articolo dal titolo *Combining self-organisation and autonomic computing in cass with aggregate-mape*[16], che consiste nella guida dinamica dei pedoni nella città di Vienna. I risultati sono riportati nel grafico in fig. 3.1.

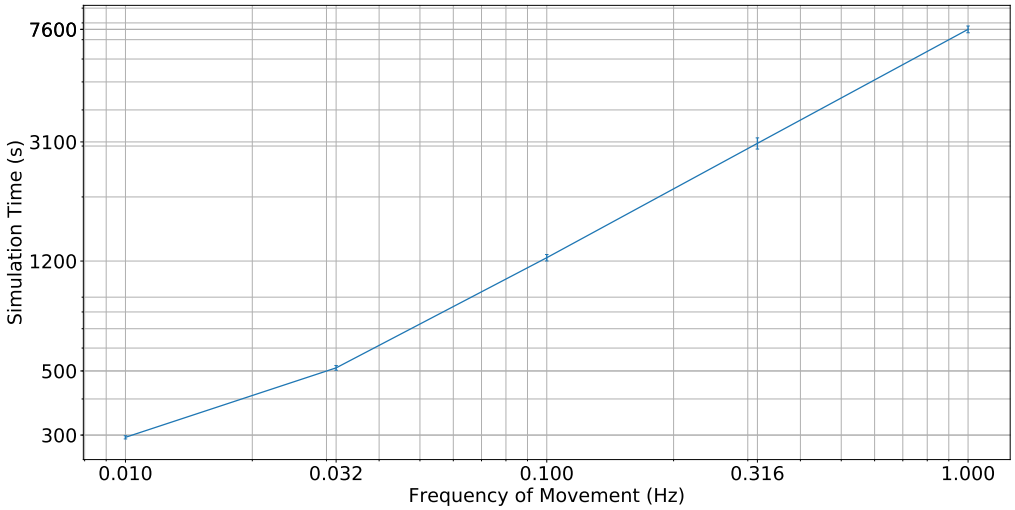


Figura 3.1: In figura è mostrato come cambia il tempo di esecuzione della simulazione in [16] al variare della frequenza con cui vengono ricalcolati i percorsi che devono essere seguiti dai pedoni. Si noti che il tempo di esecuzione cresce in maniera sostanzialmente lineare con l'aumentare della frequenza di ricalcolo. Le misure sono state ottenute facendo la media tra i tempi di esecuzione di cinque simulazioni distinte della durata di 300 unità temporali simulate ciascuna, ed è stata anche calcolata la relativa deviazione standard, rappresentata in figura dalle barre d'errore.

Dal grafico si può dedurre che, all'aumentare della frequenza di movimento dei nodi, aumenta in maniera circa lineare anche il tempo necessario al completamento della simulazione. Questo, almeno in apparenza, è normale, poichè significa che, al raddoppiare delle operazioni di routing da eseguire, raddoppia anche il tempo necessario al loro completamento, e questo è in linea con la logica di funzionamento di GraphHopper, che esegue una nuova operazione totalmente scorrelata dalle altre ogni volta che gli viene richiesto, ma non con la presenza della cache, che dovrebbe far fronte all'aumento delle richieste di nuove route con un aumento del numero di cache hit. Ciò che quindi dovrebbe far fronte all'intrinseca complessità del calcolo dei percorsi tramite GraphHopper non sta quindi funzionando come dovrebbe. Il passo successivo consiste quindi nel misurare il valore di cache hit per valutare l'efficacia della cache; di seguito viene riportato il risultato di questa misurazione.

Hit rate della cache al variare della frequenza di calcolo dei percorsi

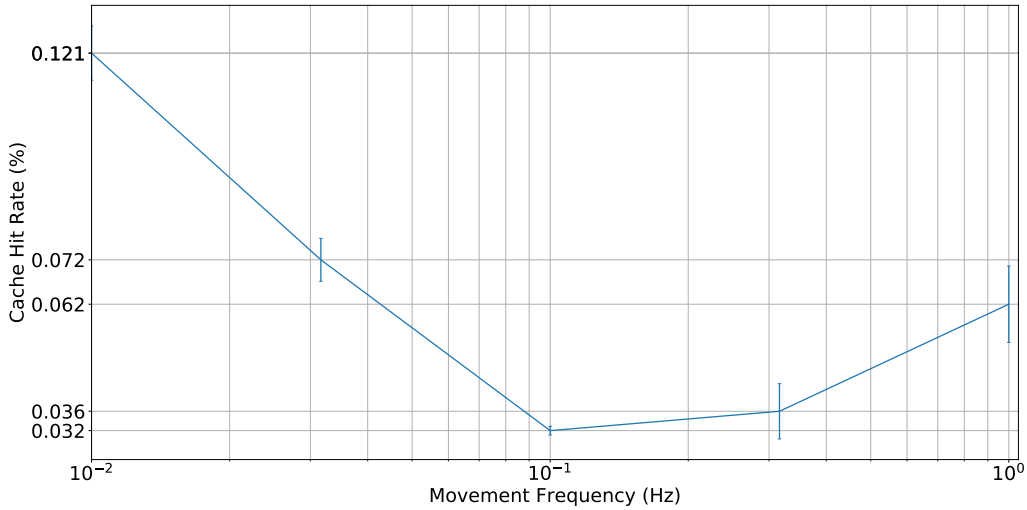


Figura 3.2: In figura è mostrato lo hit rate della cache interna ad OSMEnvironment al variare della frequenza di ricalcolo dei percorsi dei nodi nella simulazione in [16]. Si noti che, nel caso migliore, lo hit rate è poco al di sopra del 10%, chiaramente non sufficiente a rendere tangibile il guadagno prestazionale che la presenza della cache dovrebbe offrire. Le misure sono state ottenute facendo la media tra i tempi di esecuzione di cinque simulazioni distinte della durata di 300 unità temporali simulate ciascuna, ed è stata anche calcolata la relativa deviazione standard, rappresentata in figura dalle barre d'errore.

Il risultato ottenuto da questa misurazione è abbastanza deludente: anche nel migliore dei casi, lo hit rate supera a malapena il 10%, e in ogni caso si tratta solo di un picco isolato in un contesto in cui la media è molto più bassa, intorno al 5%. Questi valori sono enormemente al di sotto di quelli che renderebbe la presenza della cache produttiva e valevole del costo della sua gestione: per quanto in letteratura sia difficile trovare un'indicazione lapidaria di quale possa essere considerato un buon valore di cache hit rate, certamente non si può trattare di un valore così basso, e le misurazioni precedenti lo dimostrano.

Si pone a questo punto la questione di come migliorare le prestazioni sotto questo punto di vista. Certamente si deve risolvere il problema che affligge la cache, ma non è altrettanto immediato individuare la strategia migliore per il conseguimento di questo scopo. Questo sarà infatti il fulcro del capitolo successivo, in cui verranno illustrate le

soluzioni messe in campo per cercare di far fronte al problema e i relativi risultati.

3.2 Protelis e routing

L'argomento del routing all'interno di Alchemist non si esaurisce unicamente in relazione al suo utilizzo negli ambienti, ma prosegue nell'ambito dell'incarnazione Protelis. Come visto in precedenza, infatti, anche nell'ambito del linguaggio Protelis è possibile determinare la distanza che separa due nodi seguendo le strade sulla mappa. Per determinare questa distanza è necessario ricorrere nuovamente a GraphHopper. La frequenza con cui si valutano le distanze è indipendente da quella con cui si calcolano i movimenti dei nodi sulla mappa.

Quando il calcolo di una route viene richiesta da un programma Protelis, questa viene evasa dal contesto di esecuzione, che a sua volta si occupa di invocare GraphHopper per ottenere il risultato richiesto. A differenza di quanto avviene nell'ambiente, in questo caso non è presente una cache delle route; bisogna infatti ricordare che, mentre l'ambiente è unico per tutti i nodi che vi vivono all'interno, è presente una diversa istanza del contesto d'esecuzione per ogni nodo, perciò una cache a livello di singolo oggetto come quella dell'ambiente sarebbe stata in ogni caso poco utile, perchè non permetterebbe di usare nuovamente i risultati ottenuti grazie alle richieste di altri Execution Context.

È stato eseguito un benchmark relativamente alle prestazioni del calcolo delle route dal contesto di esecuzione; non deve sorprendere che i risultati siano analoghi a quelli ottenuti nell'ambiente.

Tempo di esecuzione al variare della frequenza del calcolo delle distanze in Protelis

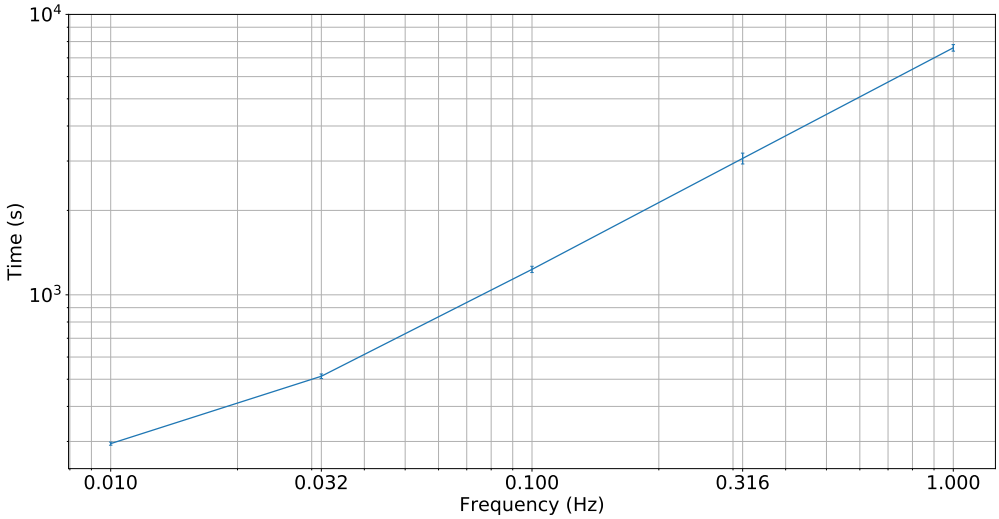


Figura 3.3: In figura è mostrato il tempo di esecuzione della simulazione in [16] al variare della frequenza di calcolo delle distanze in Protelis. Anche in questo caso, si nota che il tempo di simulazione aumenta in maniera sostanzialmente lineare all'aumentare della frequenza. Le misure sono state ottenute facendo la media tra i tempi di esecuzione di cinque simulazioni distinte della durata di 300 unità temporali simulate ciascuna, ed è stata anche calcolata la relativa deviazione standard, rappresentata in figura dalle barre d'errore.

Anche in questo caso, il tempo richiesto al completamento della simulazione aumenta linearmente con l'aumentare della frequenza con cui viene richiesto il calcolo dei percorsi. È bene ribadire che la frequenza del ricalcolo dei percorsi nell'ambiente è indipendente da quella relativa a Protelis, perciò la frequenza riportata nel grafico soprastante non è correlata con quella riportata nel grafico nella sezione precedente.

In questo caso, non essendoci una cache su cui poter lavorare per migliorare le prestazioni, è necessario pensare ad una strategia alternativa: la soluzione individuata sarà illustrata nel capitolo 4.

3.3 Quali possibili miglioramenti?

Come anticipato, viste le misurazioni effettuate, è necessario lavorare principalmente su due fronti: per quanto riguarda il calcolo delle route nell'ambiente, è necessario innanzitutto cercare di sopperire al deficit prestazionale che affligge la cache, che non è nelle condizioni di esprimere il suo potenziale, mentre per quanto riguarda il calcolo delle route dal contesto di esecuzione di Protelis, non avendo a disposizione la cache, è necessario o rendere più rapido il calcolo delle route, lavorando perciò direttamente all'interazione con la libreria che gestisce questa operazione, oppure implementare una cache o qualcosa di analogo che renda più rapido il calcolo delle route senza dover metter mano a GraphHopper.

Nel particolare caso dell'ambiente, i miglioramenti si prospettano sostanziosi: allo stato attuale delle cose, il cache hit rate è talmente basso che un eventuale suo miglioramento potrebbe portare ad un incremento elevato e immediatamente tangibile delle prestazioni; naturalmente, la difficoltà sta nell'individuare quale sia un modo adatto per ottenere questo risultato, e questo sarà l'oggetto del prossimo capitolo.

4

Miglioramento delle performance di Alchemist

In questo capitolo verranno discusse le varie soluzioni implementate al fine di migliorare la velocità con cui Alchemist è in grado di ottenere il percorso che collega due punti sulla mappa. Come verrà poi descritto, sono state messe in campo diverse soluzioni che concorrono al miglioramento delle prestazioni, la cui efficacia verrà successivamente discussa.

4.1 Ottimizzazione di routing e caching in OSMEnvironment

Notando, come illustrato nel capitolo precedente, le scarse prestazioni della cache interna ad OSMEnvironment, si è cercato di capire quale potesse essere la strategia migliore per aumentare il cache hit rate. La classe LoadingCache della libreria Google Guava, utilizzata per implementare la cache all'interno di OSMEnvironment, annovera tra le sue funzionalità anche la eviction automatica delle entry a cui non si accede da un certo lasso di tempo, impostabile manualmente. Pur ottenendo un cache hit rate molto basso, la cache risultava sempre piena di route calcolate precedentemente, che però non venivano utilizzate, se non raramente, e venivano di conseguenza eliminate dalla cache.

Dopo alcuni esperimenti, è stato individuato il motivo di queste scarse prestazioni. All'interno della cache le entry consistono in oggetti Java che identificano una route, e sono considerati uguali quando sono uguali il punto di partenza, il punto di arrivo e il

veicolo utilizzato. Inoltre, per reperire le entry all'interno della cache, viene utilizzato il suo hash code, in quanto, a livello fisico, LoadingCache gestisce i dati che contiene mediante un'organizzazione hash (infatti la sua implementazione è basata su quella di `ConcurrentHashMap`). Le chiavi sono perciò degli oggetti che identificano i percorsi, mentre i valori associati alle chiavi sono i percorsi stessi. Ciò che accade nel momento in cui è necessario ottenere una route è perciò riassumibile come segue:

1. Si cerca all'interno della cache una entry consistente in una route che abbia stesso punto di arrivo, lo stesso punto di partenza e lo stesso veicolo

2. Se questa è presente, viene restituita, altrimenti

3. Si richiede a GraphHopper la route desiderata, che viene poi inserita nella cache e restituita

Il problema sta nel primo punto: poichè un punto sulla mappa è identificato dalle sue coordinate, che possono avere una risoluzione anche molto elevata, è necessario, affinchè una route possa essere reperita nella cache, che quella richiesta coincida con la stessa elevata precisione con una di quelle già presenti, sia per quanto riguarda il punto di origine che quello di arrivo, che sono gli identificatori dei percorsi nella cache. Questo può portare a situazioni paradossali: se una route presente in cache differisce nel punto di partenza o in quello di arrivo anche per un solo millimetro rispetto a quella richiesta, è necessario calcolarne una totalmente nuova, in quanto, paragonando i punti di partenza e di arrivo sulla mappa, questi risultano di fatto differenti. Questo spiega il basso hit rate: è chiaramente raro, seppur non impossibile, che venga richiesta due volte la stessa route con gli stessi identici punti di partenza e arrivo, mentre è certamente più frequente che siano richiesti i percorsi tra punti vicini tra loro sulla mappa, anche se non uguali, ma con questa logica di funzionamento quest'ultima tipologia di richiesta non riceve alcun incremento prestazionale dalla presenza della cache.

Occorre quindi cambiare il modo in cui la cache restituisce i percorsi già calcolati, consentendo anche l'ottenimento di percorsi che, nonostante non siano esattamente identici a quello richiesto, siano comunque sufficientemente simili da poter essere considerati sostanzialmente uguali. Quale sia il livello di somiglianza oltre il quale è possibile considerare due percorsi uguali non è determinabile a priori, ma dipende piuttosto dalle caratteristiche della simulazione: è chiaramente diverso avere una simulazione in cui si

hanno, ad esempio, dei pedoni che si muovo all'interno di un piccolo giardino da una in cui si hanno dei camion che percorrono centinaia di chilometri sull'autostrada.

Dal momento che, come già detto, le chiavi di una LoadingCache sono oggetti Java contenenti il punto di partenza e il punto di arrivo di un percorso, è stata definita una nuova classe i cui oggetti saranno utilizzati come chiavi della LoadingCache. Questa classe, oltre naturalmente al punto di partenza e a quello di arrivo 'originali', contiene anche delle loro versioni approssimate. Mentre, chiaramente, i punti originali sono memorizzati così come vengono passati al costruttore, i valori approssimati sono ottenuti grazie a questo metodo:

```
1 private double approximate(final double value) {
2     return Double.longBitsToDouble(Double.doubleToLongBits(
          value) & (0xFFFFFFFFFFFFFFFFL << appr));
3 }
```

Il valore originale della coordinata viene approssimato mettendo il suo valore espresso come numero floating point a doppia precisione secondo la codifica IEEE 754 in formato binario[1] in AND con una maschera di bit la cui lunghezza è determinabile dall'utente in base alle sue necessità, modificando un apposito parametro nel file YAML della simulazione. La scelta di utilizzare un approccio di così basso livello è stata dettata dalla volontà di introdurre un overhead il più limitato possibile, implementando queste modifiche alla cache in maniera leggera.

A questo punto è stato necessario ridefinire anche i metodi `hashCode` e `equals` nel modo seguente:

```
1 @Override
2 public int hashCode() {
3     if (hash == 0) {
4         hash = HashUtils.hash32(v, apprStart, apprEnd);
5     }
6     return hash;
7 }
```

```
1 @Override
2 public boolean equals(final Object obj) {
3     if (obj instanceof OSMEnvironment.CacheEntry) {
```

```
4            final OSMEnvironment<?>.CacheEntry other = (
                 OSMEnvironment<?>.CacheEntry) obj;
5            return v.equals(other.v) && apprStart.equals(other.
                 apprStart) && apprEnd.equals(other.apprEnd);
6        }
7        return false;
8    }
```

Come si può notare, il cacolo dello hash code, ottenuto tramite una funzione di libreria che fa uso dell'algoritmo Murmur3-32bit[2], è basato non più sulle posizioni esatte, ma su quelle approssimate; analogamente avviene nel metodo `equals`.

Naturalmente rimane possibile utilizzare il sistema preesistente basato sulle posizioni esatte nel caso lo si desideri.

Sono stati svolti dei benchmark con lo scopo di misurare l'impatto prestazionale di questa soluzione, i cui risultati sono riportati in fig. 4.1.

Tempo di esecuzione al variare della frequenza di calcolo dei percorsi dopo la modifica alla cache di OSMEnvironment

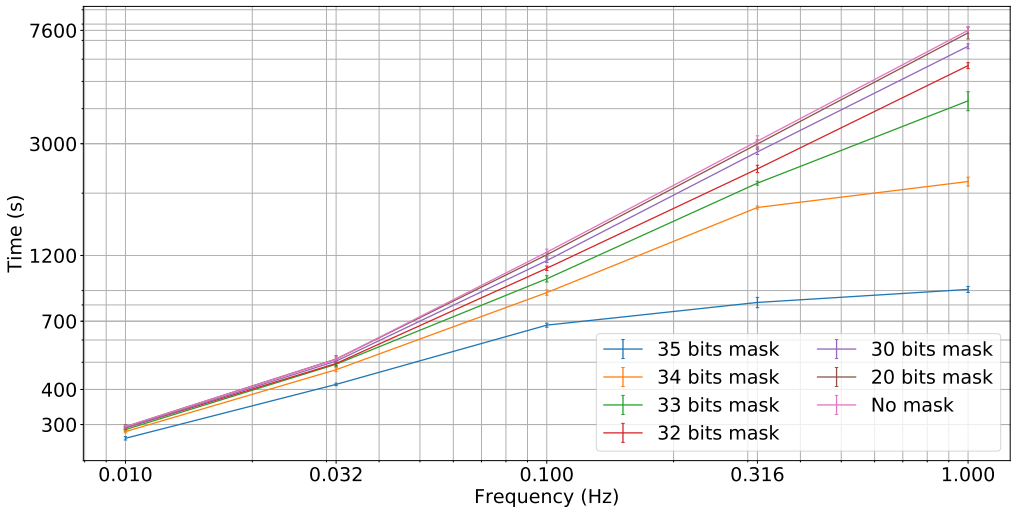


Figura 4.1: In figura è mostrato il tempo di esecuzione della simulazione in [16] al variare della frequenza di calcolo dei percorsi, misurato dopo aver fatto le modifiche alle cache descritte nel capitolo 4.1 . È evidente che, man mano che si aumenta la dimensione della maschera, e di conseguenza il livello di approssimazione nel reperimento dei percorsi nella cache, il tempo di esecuzione diminuisce rispetto a quello ottenuto senza alcuna approssimazione. Le misure sono state ottenute facendo la media tra i tempi di esecuzione di cinque simulazioni distinte della durata di 300 unità temporali simulate ciascuna, ed è stata anche calcolata la relativa deviazione standard, rappresentata in figura dalle barre d'errore.

Come si può notare, aumentando l'approssimazione ammissibile, aumenta anche la velocità della simulazione, arrivando ad una riduzione del tempo di esecuzione di 10 volte mascherando gli ultimi 35 bit di ogni coordinata. Anche con approssimazioni più moderate, comunque, l'incremento prestazionale c'è ed è evidente, ed è imputabile all'aumento del cache hit rate, come si può vedere dal grafico in fig. 4.2. È inoltre da notare che, aumentando la lunghezza della maschera di bit, il tempo di esecuzione cresce in maniera non più lineare con l'aumentare della frequenza di calcolo dei percorsi, ma tende piuttosto ad avvicinarsi ad essere logaritmico, e questo non può che essere un fattore molto positivo.

Hit rate della cache al variare della frequenza di calcolo dei percorsi dopo la modifica alla cache di OSMEnvironment

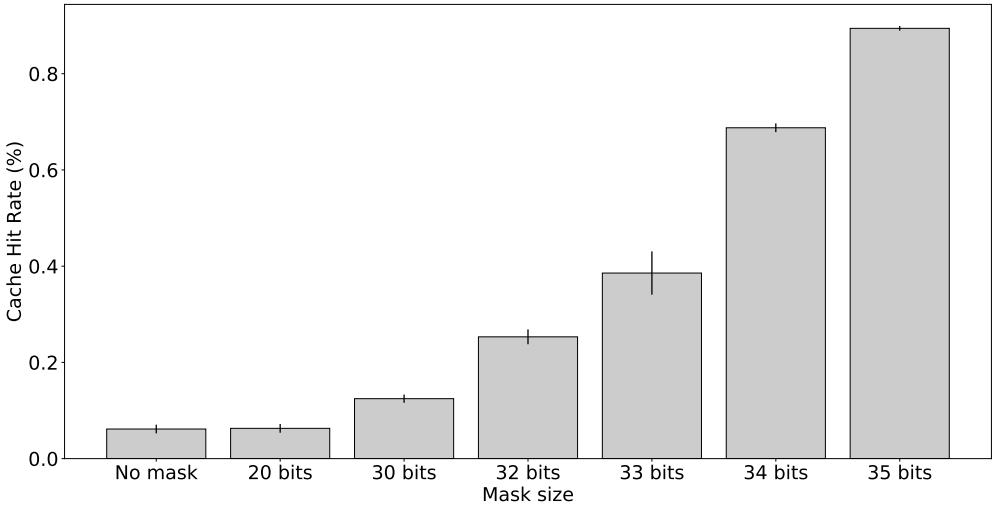


Figura 4.2: In figura è mostrato lo hit rate della cache interna ad OSMEnvironment al variare della frequenza di ricalcolo dei percorsi dei nodi nella simulazione in [16]. Si noti che, nel caso migliore, lo hit rate supera il 90%, mostrando chiaramente l'efficacia della modifica alla cache, che prima arrivava solamente intorno a 5-10% di hit rate, come si può vedere con più precisione in fig. 3.2. Le misure sono state ottenute facendo la media tra i tempi di esecuzione di cinque simulazioni distinte della durata di 300 unità temporali simulate ciascuna, ed è stata anche calcolata la relativa deviazione standard, rappresentata in figura dalle barre d'errore.

Si può vedere chiaramente che, aumentando la dimensione della maschera di bit, si passa dallo scarso 10% di hit rate iniziale a oltre il 90% con una maschera di 35 bit, segno inequivocabile che il sistema funziona e funziona bene.

Può ora sorgere spontanea una domanda: a che approssimazione corrisponde sulla mappa una certa maschera di bit? La risposta in realtà non è immediata: bisogna infatti considerare che la terra è (grossomodo) sferica, perciò, a seconda della latitudine a cui ci si trova, angoli diversi corrispondono a lunghezze diverse sulla superficie. Ad ogni modo, in prima approssimazione, si può considerare che con 32 bit di maschera equivalgono, nella simulazione [16], ad un'approssimazione di circa 500 metri, e ogni volta che si riduce di un bit la lunghezza della maschera questo valore si dimezza, in accordo con il funzionamento della codifica IEEE 754.

4.2 Miglioramenti all'ExecutionContext di Protelis

La questione delle prestazioni del routing si pone, come già visto, anche nel contesto dell'incarnazione Protelis, e più nello specifico all'interno del contesto d'esecuzione. In questo caso non è presente una cache su cui poter lavorare per poter migliorare le performance, perciò è necessario agire su altri fronti. Come già accennato in precedenza, all'interno dell'Execution Context il calcolo dei percorsi viene utilizzato nel determinare la distanza di un nodo dagli altri, e questa operazione viene fatta con una certa frequenza definibile dall'utente. Ogni volta che si vuole, ad esempio, sapere la distanza che separa un nodo dagli altri seguendo le strade, è necessario invocare GraphHopper per l'ottenimento di questo dato. Naturalmente, come nel caso del calcolo dei percorsi dei nodi sulla mappa, questa operazione è costosa, perciò è stato modificato il metodo del contesto d'esecuzione che si occupa di gestire queste richieste: ogni volta che viene ricevuta una richiesta di questo tipo, oltre a fornire il dato, questo viene memorizzato in una variabile, e tutte le volte che questa informazione verrà richiesta nuovamente verrà restituito il valore precalcolato, a meno che non sia passato un intervallo temporale sufficientemente grande, configurabile dall'utente.

In questo modo si evita di ricalcolare spesso un dato che cambia in maniera significativa solo dopo un certo intervallo di tempo. Anche in questo caso, come in quello precedente, spetta all'utente determinare quali intervalli temporali siano da considerare accettabili e quali invece no, a seconda delle caratteristiche della simulazione.

Anche in questo caso sono stati eseguiti dei benchmark sulla simulazione di prova, e i risultati sono mostrati in fig. 4.3.

Tempo di esecuzione al variare della durata della validità della distanza precalcolata

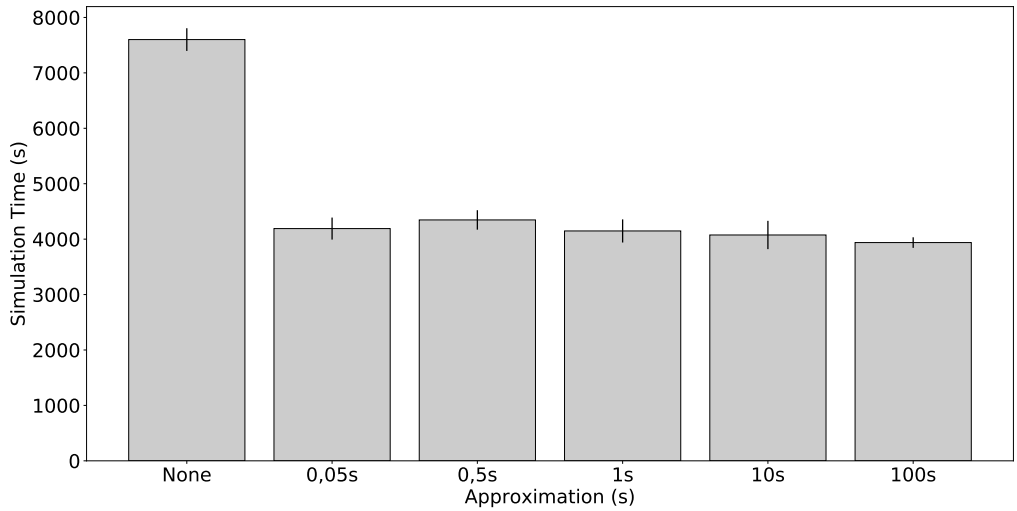


Figura 4.3: In figura è rappresentato il tempo di esecuzione della simulazione [16] al variare dell'intervallo temporale che intercorre tra due effettivi ricalcoli della distanza tra due nodi all'interno del contesto di esecuzione di Protelis. Si noti come applicare questa approssimazione costituisca un grande vantaggio in termini di tempo di esecuzione, ma al contempo approssimazioni diverse comportano un guadagno prestazionale sostanzialmente identico. Questo può essere imputabile al fatto che, una volta ottenuto un certo guadagno in termini di velocità, il collo di bottiglia si sposta altrove all'interno dell'interprete Protelis e/o dell'incarnazione Protelis di Alchemist.

Come si può notare, rispetto alla simulazione eseguita senza alcuna approssimazione, i tempi di esecuzione che si ottengono impiegando questa nuova soluzione sono significativamente più bassi. C'è d'altra parte anche un aspetto un po' imprevisto: sarebbe infatti logico aspettarsi che, man mano che l'approssimazione aumenta, diminuisca il tempo di esecuzione, mentre, al netto dell'errore, tutte le misure sono pressochè identiche a prescindere dal livello di approssimazione adottato. Questo può essere attribuibile al fatto che, velocizzando l'elaborazione all'interno del contesto di esecuzione, il collo di bottiglia si sposta da un'altra parte del programma, possibilmente all'interno dell'interprete Protelis, che comunque deve continuare ad eseguire il programma che ha causato l'invocazione del calcolo della distanza.

Ad ogni modo, l'intervento all'Execution Context ha riguardato il calcolo delle distanze tra i nodi, mentre c'è ancora margine di miglioramento per quanto riguarda il calcolo dei percorsi.

4.3 Discussione

Senz'altro l'impatto prestazionale dell'utilizzo di GraphHopper, e del routing in generale, risulta essere elevato, perciò le soluzioni presentate, per quanto frutto di un compromesso tra velocità di esecuzione e qualità del risultato, visto il guadagno che permettono di ottenere in termini di tempo di calcolo, sono apprezzabili e certamente utili. Bisogna comunque considerare quanto possa essere accettabile la perdita di precisione nel risultato quando si adotta l'approssimazione nel calcolo dei percorsi sulla mappa, che può essere anche nell'ordine delle centinaia di metri, se non dei chilometri, e questo necessariamente varia in base a cosa si sta simulando. Un discorso simile si applica all'uso dell'approssimazione nel calcolo delle distanze del vicinato nel contesto di esecuzione Protelis, anche se in questo caso non bisogna considerare l'approssimazione in termini di distanza, ma piuttosto in termini di frequenza di aggiornamento, che, per essere valutata nel suo impatto sul risultato, deve essere messa in relazione alla frequenza e alla velocità degli eventi nella simulazione: anche in questo caso, perciò, non si può fare un discorso generale, ma bisogna considerare caso per caso le singole simulazioni.

In sintesi, il miglioramento fornito da queste soluzioni c'è ed può essere anche sostanzioso, e spetta a chi scrive una simulazione valutare se è disposto ad accettare i compromessi che queste comportano.

5
Conclusioni e lavori futuri

In base a quanto visto nei capitoli precedenti, l'intento di migliorare le prestazioni nell'ambito delle simulazioni che fanno ricorso alle funzionalità di routing, e quindi della libreria GraphHopper, è da considerarsi soddisfatto, pur nei limiti già descritti nelle discussioni delle particolari soluzioni tecniche adottate. Certamente comunque c'è ancora un ampio margine di miglioramento. In primo luogo, infatti, il tuning del routing effettuato per mezzo dell'approssimazione mediante maschera di bit, è tanto efficace dal punto di vista prestazionale quanto difficile da usare per un utente che vuole fare di Alchemist unicamente uno strumento di lavoro senza possedere particolari competenze informatiche, in quanto si tratta di un approccio di basso livello e, di conseguenza, di non immediata compresione. In secondo luogo, c'è ancora la possibilità di aumentare la velocità di esecuzione delle simulazioni ripensando non solo la logica di funzionamento della parte del contesto di esecuzione di Protelis che si occupa di valutare la distanza tra i nodi, ma anche di quella parte che si occupa di valutare quali nodi sono effettivamente da prendere in considerazione.

Ad ogni modo, al netto di queste considerazioni, non è fuori luogo affermare che queste modifiche rappresentano per Alchemist un ulteriore salto in avanti nella diffusone in ambito accademico: la possibilità di ottenere rapidamente dei risultati, seppur approssimati, è sicuramente importante quando l'intento è quello di simulare degli eventi la cui rappresentazione può essere complessa e quindi bisognosa di alcuni tentativi prima del raggiungimento della forma finale, poichè permette di testare rapidamente un modello verificandone, almeno in maniera approssimativa, l'attendibilità. Inoltre, può essere utile anche semplicemente in quei casi in cui la massima precisione del risultato di una simulazione è gradita ma non necessaria, prediligendo in questo modo la velocità di esecuzione

sulla precisione. In aggiunta, può essere utile anche a scopo dimostrativo, nell'illustrare in tempo reale le potenzialità di una certa simulazione o di Alchemist stesso.

In questa tesi abbiamo visto alcuni aspetti del funzionamento di Alchemist e il relativo impatto dal punto di vista prestazionale, concentrandoci in particolare sull'impatto che il calcolo dei percorsi su una mappa ha sul tempo di esecuzione di una simulazione. Ci si è perciò successivamente concretati sull'individuazione di un metodo che consentisse di migliorare le prestazioni in questo particolare contesto, arrivando ad una soluzione che introduce la possibilità di aumentare la velocità di esecuzione a discapito della precisione del risultato finale, in modo da permettere di terminare più rapidamente la simulazione nel caso in cui non sia richiesta la massima precisione possibile. Sono stati anche spiegati i limiti in tal senso, come l'entità della perdita di precisione.
Alchemist è un software complesso, con una codebase di oltre 50000 linee di codice e oltre 100 librerie impiegate, la cui modifica è tutt'altro che immediata, perciò il raggiungimento di questo obiettivo è da considerare un buon risultato, pur con le considerazioni che sono state oggetto di questa trattazione.

Bibliografia

[1] IEEE standard for floating-point arithmetic.

[2] Austin Appleby. Murmur3 hash function. *Available from: https://code.google.com/p/smhasher/[accessed 1 May 2015].*

[3] Eduard Babulak and Ming Wang. Discrete event simulation: State of the art. *International Journal of Online Engineering*, 4(2), 2008.

[4] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[5] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[6] Bill Bejeck. *Getting Started with Google Guava*. Packt Publishing Ltd, 2013.

[7] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *yaml. org, Tech. Rep*, 2005.

[8] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[9] George S Fishman. Principles of discrete event simulation.[book review]. 1978.

[10] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, mar 2000.

[11] Sara Montagna, Andrea Omicini, and Danilo Pianini. Extending the gillespie's stochastic simulation algorithm for integrating discrete-event and multi-agent based simulation. In *Multi-Agent-Based Simulation XVI - International Workshop, MABS 2015, Istanbul, Turkey, May 5, 2015, Revised Selected Papers*, pages 3–18, 2015.

[12] Sara Montagna, Andrea Omicini, and Danilo Pianini. A gillespie-based computational model for integrating event-driven and multi-agent based simulation: Extended abstract. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*, pages 1763–1764, 2015.

[13] Sara Montagna, Danilo Pianini, and Mirko Viroli. A model for drosophila melanogaster development from a single cell to stripe pattern formation. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1406–1412, 2012.

[14] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, jan 2013.

[15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1846–1853. ACM, 2015.

[16] Mirko Viroli, Antonio Bucchiarone, Danilo Pianini, and Jacob Beal. Combining self-organisation and autonomic computing in cass with aggregate-mape. In *Foundations and Applications of Self* Systems, IEEE International Workshops on*, pages 186–191. IEEE, 2016.

[17] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *ESOCC Workshops*, volume 393, pages 114–128, 2013.

[18] Mirko Viroli, Danilo Pianini, Sara Montagna, Graeme Stevenson, and Franco Zambonelli. A coordination model of pervasive service ecosystems. *Sci. Comput. Program.*, 110:3–22, 2015.

Ringraziamenti

Vorrei ringraziare la mia famiglia per tutto il supporto che mi ha offerto in questi anni, aiutandomi a migliorare e a dare sempre il massimo: non avrei potuto farcela senza di voi.

Vorrei inoltre ringraziare Danilo, senza il quale questi mesi di lavoro non sarebbero certamente stati gli stessi, e senza il quale forse ora non sarei qui a scrivere questa tesi.